SEUPD@CLEF: Team kueri on Argument Retrieval for Comparative Questions

Maria Aba[1], Munzer Azra[1], Marco Gallo[1], Odai Mohammad[1], Ivan Piacere[1], Giacomo Virginio[1] and Nicola Ferro[1]

[1]*University of Padua, Italy*

Abstract

In this paper we present the information retrieval system we developed for the 2022 Touché @ CLEF Task 2 evaluation campaign. The participation in the task is performed as a student group project conducted in the Search Engines course a.y. 2021/2022 at the Computer Engineering and Data Science master degrees at University of Padua.

This tasks' aim is to create systems that are able to retrieve documents that compare two options, e.g. which is the best pet between a dog and a cat.

Here we describe the architecture of our system, we list the software and hardware resources we made use of, we discuss the results obtained using different configurations and finally we present improvements which could be applied to our system to enhance its performance.

Keywords

Information retrieval, Comparative questions, Lucene

1. Introduction

Before the era of the internet, information storage and retrieval systems were mostly used by professionals for medical research, in libraries, by governmental organizations, and archives. Therefore, access to such information was a hard process especially for non-search experts. Recently, with the fast increase in the number of data and information available online, the importance of search engines grew rapidly. Nowadays, people use search engines to locate and buy goods, choose a vacation destination, select a medical treatment, etc. Search engines transitioned from being searchers' tools for information to tools for building opinions and making major decisions. All of these aspects, when considered together, make retrieval systems a need for impacting the industry and improving the field of information retrieval.

This paper is structured as follows: Section 2 presents related work; Section 3 describes our approach; Section 4 explains our experimental setup; Section 5 discusses our main findings; finally, Section 6 draws some conclusions and outlooks for future work.

CLEF 2022: Conference and Labs of the Evaluation Forum, September 5–8, 2022, Bologna, Italy

maria.aba@studenti.unipd.it (M. Aba); munzer.azra@studenti.unipd.it (M. Azra); marco.gallo.9@studenti.unipd.it (M. Gallo); odai.mohammad@studenti.unipd.it (O. Mohammad); ivan.piacere@studenti.unipd.it (I. Piacere); giacomo.virginio@studenti.unipd.it (G. Virginio); ferro@dei.unipd.it (N. Ferro)

http://www.dei.unipd.it/~ferro/ (N. Ferro)

0000-0001-9219-6239 (N. Ferro)

2. Related Work

The packages 3.1, 3.2, 3.3 and 3.4 are obtained expanding from a baseline built on the TIPSTER collection, during lessons of the "Search Engines" course, University of Padua.
Information about the course can be found at the following link:
https://en.didattica.unipd.it/off/2021/LM/IN/IN2547/004PD/INQ0091599/N0

3. Methodology

The following is the class diagram for our implementation: Figure 1

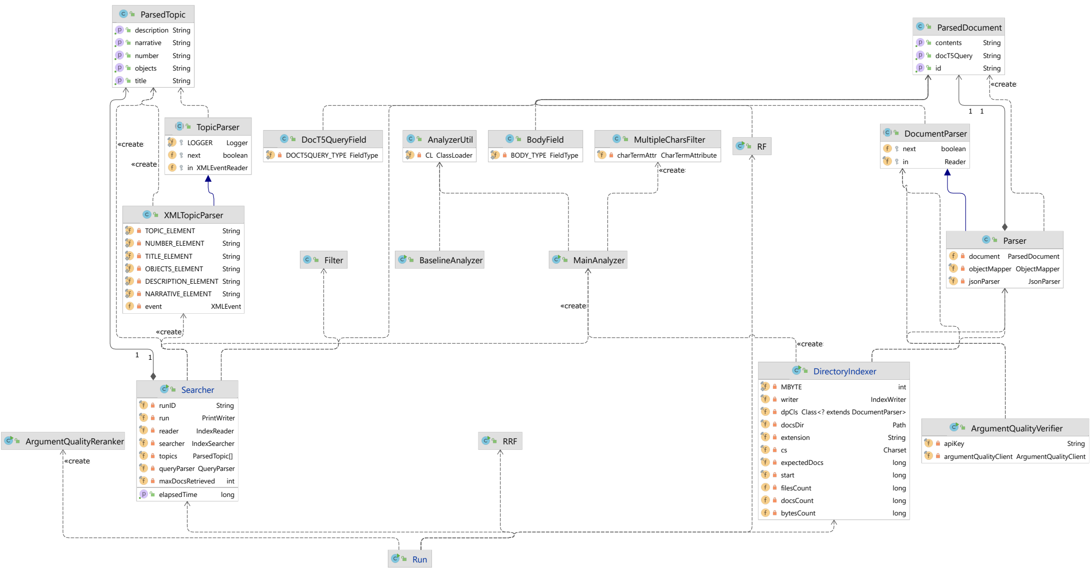

Figure 1: Class diagram of the project

The developed Java system is divided into the following packages, each package representing a stage:

3.1 Parse

This package is divided into two packages:

3.1.1 Document

The aim of this package of the project is to facilitate the parsing of the document corpus provided by CLEF for the Touchè Task 2 so that they can be used together with Lucene. The corpus for Task 2 is a collection of about 0.9 million text passages contained in a single JSON file passages.json. The file contains several documents organized as nodes of a JSON tree and each node contains 3 different fields. Namely id, contents, and chatNoirUrl. In this project only the id and the contents data of the document are retrieved from the JSON node and used in the implementation. The chatNoirUrl is not taken into account since we found it to not be relevant for the task. Furthermore, CLEF also provided a version of the corpus with text passages expanded with queries generated using DocT5Query. We found this expanded version valuable and added support for parsing it in our implementation. The DocT5Query queries are contained within the 'contents' key of each document. The parsing is implemented in the following classes:

1. DocumentParser.java: The class creates a document parser that takes a reader object to be used as input, it overrides the hasNext() and next() methods, and performs the actual parsing.
2. Parser.java: This class extends the aforementioned DocumentParser class and takes the reader object as input. The hasNext() method has been implemented and it is where the actual parsing takes place and it extracts the three already mentioned fields.
3. ParsedDocument.java: Represents the actual document to be indexed by Lucene. It defines the id, contents, and docT5Query of the JSON documents. The class defines proper getter and setter methods for the various fields to be easily retrieved and set, and it overrides utility methods like toString(), equals(), and hashCode().

3.1.2 Topic

The aim of this package of the project is to facilitate the parsing of the topics provided by CLEF for the Touchè Task 2 so that they can be used together with Lucene. The topics of Task 2 is a collection of 50 topics contained in a single XML file topics-task2.xml. The file contains the topics each having 5 different attributes. Namely number, title, objects, description, and narrative. In this project, even though all the attributes of a topic are parsed, Only the number, objects, and title are used for search. The narrative and description attributes are used for manual relevance. The parsing is implemented in the following classes:

1. TopicParser.java: The class creates a topic parser that takes a reader object to be used as input, it overrides the hasNext() and next() methods, and performs the actual parsing.
2. XMLTopicParser.java: This class that extends the aforementioned TopicParser class and takes the reader object as input. The hasNext() method has been implemented and it is where the actual parsing takes place and it extracts the already mentioned fields.
3. ParsedTopic.java: Represents the actual topic to be used in the searcher with Lucene. It defines all the attributes of a topic. The class also defines proper getter and setter methods for the various fields to be easily retrieved and set, and it overrides utility methods like toString(), equals(), and hashCode().

3.2 Analyze

We have built three analyzers, to process the documents, perform tokenization and use different combination of filters.

3.2.1 AnalyzerUtil.java

This is a helper class auxiliary class containing utility methods for loading stop lists in the resource folder. These stop lists were obtained from well-known standard lists based off high frequency words. Some of them are: Atire, Indri, Smart, Terrier, Zettair, Glasgow, Snowball, Okapi, and Lucene.

3.2.2 BaselineAnalyzer.java

This is a java class for tokenization which starts with a StandardTokenizer and reduces every word to lower case using a LowerCaseFilter and then we have used the StopFilter to remove frequent words in the collection that do not bring useful information.

3.2.3 MainAnalyzer.java

Contains various filters from AnalyzerUtil as well as EnglishPossessiveFilter and MultipleChars-Filter. It also adds synonyms dictionary to perform a query expansion based on Wordnet.

3.3 Index

This package contains the 3 classes responsible for the index creation, they are as follows:

3.3.1 BodyField.java

Represents the body of a specific document it has two different constructors, one accept a Reader and the other accept a String value. The only field is BODY_TYPE which is tokenized and not stored, keeping only document ids and term frequencies in order to minimize the space occupation.

3.3.2 DirectoryIndexer.java

It's used for indexing the whole directory tree, it takes as parameters the Analyzer to be used, the Similarity, the size in megabytes of the buffer for indexing documents, the directory where to store the index, the directory from which documents have to be read, the extension of the files to be indexed, the charset used for encoding documents, the total number of documents expected to be indexed and the class of the DocumentParser to be used. The constructor handles several exceptions that may rise and takes care of the index writer configuration. For testing purposes we added a main method which creates a new DirectoryIndexer using custom parameters and then runs the method index which does the actual indexing of the documents and skips every file which doesn't have the correct extension. It's important to note the fact that we added a new custom parameter DocT5QueryField.

3.3.3 DocT5QueryField.java

It manages the new tokenized and not stored field for the document, differently from the body field this class has only one constructor which accepts Strings.

3.4 Search

The search package contains just the Searcher class. This class is responsible for:

1. Retrieving and preparing the topics for the search.
   The topics are retrieved directly from the topics file and parsed using the XMLTopicParser class. Then an Analyzer is defined to tokenize and filter the tokens before the search.
2. Defining how to use topics in the search.
   We decided to use just the topics titles in the search by similarity, then we added the topic objects (the items which the user wants to compare) as a MUST clause in the search, this way we retrieved only documents documents that presented all the terms in the topic objects field. Moreover we made it possible to assign weights to the different fields of the documents among which to search, or to select just one of the two fields (Contents and DocT5Query).
3. Defining which type of comparison to perform between topics and documents.
   The searcher class accepts as a parameter a similarity function to compare the two.
4. Writing the results on a file

3.5 RF

This package contains a single class, also called RF.

RF.java is a customized class with the goal of performing a search using relevance feedback to perform query expansion.

RF functions in a similar way to the Searcher class, with the exception of building the query used in the searching using the tokens present in relevant documents, instead of using the terms in title field of the topics file.

The class collects all docID and relevance of relevant documents in the *qrels* file.

The tokens and their frequency in the relevant documents are retrieved by searching the document by docID and iterating through its termvector.

The tokens used in the search are boosted by their frequency in the document multiplied by the square of the relevance score.

Relevance Feedback is standardly based on the Rocchio Algorithm. The formula for the Rocchio Algorithm is:

$$\overrightarrow{Q_m} = \left(a \cdot \overrightarrow{Q_O}\right) + \left(b \cdot \frac{1}{|D_r|} \cdot \sum_{\overrightarrow{D_j} \in D_r} \overrightarrow{D_j}\right) - \left(c \cdot \frac{1}{|D_{nr}|} \cdot \sum_{\overrightarrow{D_k} \in D_{nr}} \overrightarrow{D_k}\right)$$

where $\overrightarrow{Q_m}$ is the modified query vector, $\overrightarrow{Q_O}$ is the original query vector, $\overrightarrow{D_i}$ is the document vector for the i^{th} document, D_r is the set of relevant documents, D_{nr} is the set of non-relevant documents and a, b and c are weight parameters.

In our case the parameters used are 0, 1, 0.

Rocchio algorithm is however defined for working with binary relevance, since this collection uses graded relevance, our version of RF is customized to take into account the different relevance scores used (0 to 3).

The custom formula we used is:

$$\overrightarrow{Q_m} = k_i \cdot \frac{1}{|D_r|} \cdot \sum_{\overrightarrow{D_i} \in D_r} \overrightarrow{D_i}$$

where $\overrightarrow{Q_m}$ is the modified query vector, $\overrightarrow{D_i}$ is the document vector for the i^{th} document, D_r is the set of relevant documents, and k_i is the relevance score of the i^{th} document.

The results of the search are then outputted as a standard run file.

3.6 RRF

This package contains a single class, also called RRF.

RRF.java is a customized class with the goal of performing using Reciprocal Ranking Fusion [1] to fuse the results of different runs in a single one.

RRF takes in imput a directory path and performs RRF using all the runs in .txt documents inside that directory.

For each documents and for each topic the documents and their respective ranking are collected.

Then document receive a new scoring using the RRF formula.

Given a set of documents *D* and a set of rankings *R* for the documents, the formula for RRF is:

$$RRFscore(d \in D) = \sum_{r \in R} \frac{1}{k + r(d)}$$

where k is a fixed number, in this case k is set to 30.

Then, for each topic, documents are ranked (and ordered) based on their RRF score.

The results of the search are then outputted as a standard run file.

3.7 Filter

The main filter class contains two methods responsible for adding the term to the search object. The main method is filterAnd. It takes two parameters as input: a query parser object to convert the string into a meaningful term for the search method. And a String s that represents the term that needed to add to the query parser. It will return an object of BooleanQuery.Builder which can be consumed later by the search method. The second method is just a helper method to extract the string object from the "objects field", tokenize the sentences, and remove any unwanted characters.

3.8 Argument quality

We decided to make use of IBM Project Debater API.

Project Debater is an AI system used to perform various tasks about debating at a human level. IBM makes freely available, for research purposes, some services based on this system through an API. [2]

We were interested in the argument quality service of the API. It accepts a couple of strings labeled as Sentence and Topic, and it returns a float score in the range 0-1 based on the relevance of the sentence for the topic and on the quality of the sentence as a text, which means how good it is written.

Since the rest of our system is designed to already score documents based on the relevance to the topic, we now just wanted to evaluate the text quality. In order to do so, we decided to send Sentence-Topic pairs in which the Topic part was an empty string.

We coded the ArgumentQualityVerifier class which evaluates the written quality of each document by using the API and then saves the scores to a file.

Then we had to use the obtained scores to rerank the results of the search saved in a run file. So we defined the ArgumentQualityReranker class which:

1. loads the quality scores of all the documents from the file into a Map object
2. iterates over the lines of the old run file and for each: multiplies the old score by the one assigned by Project Debater API and saves the object representing the new line to a list
3. sorts the list of new lines by topic number and score and writes them on a new run file

4. Experimental Setup

4.1 Collections

Some of the collections used throughout the process of system development were the ones provided by CLEF for the Touché 2022 edition. Those include:

- topics-task2.xml which contains the topics.
- The original version of passages.jsonl which contains the documents.
- DocT5Query expanded version of passages.jsonl which contains the documents expanded with queries generated using DocT5Query.

Other collections are:

- Historical stoplists: lucene, smart and terrier;
- Custom stoplists:
  - Kueristop - Stoplist formed by the 400 most concurrent term in the Contents field of the document collection;
  - Kueristopv2 - Subset of kueristop, obtained by removing from it terms appearing in the Objects field of the topics, except for the very general terms also appearing in lucene stoplist ("in" and "the").
- Sentence quality - file containing, for each document in the document Collection, the pairs of docIds and the score obtained by that document as explained in 3.8.

4.2 Evaluation measures

The evaluation measure used is Normalized Discounted Cumulative Gain at depth 5, NDCG@5 in short.

It is the evaluation measure used by Touché to officially evaluate runs.

NDCG@k is calculated as follows:

$$NDCG@k = \frac{DCG@k}{iDCG@k}$$

where

$$DCG@k = \sum_{i=1}^{k} \frac{relevance_i}{log_2(i+1)}$$

and iDCG@k is the ideal DCG@k, meaning the DCG@k for documents ordered by relevance, highest to lowest.

4.3 Git repository

The project's development can be found in the following link to its Git repository:

https://bitbucket.org/upd-dei-stud-prj/seupd2122-kueri/src/master/.

4.4 Hardware

The specifications of the computer used to perform the runs are the following:

OS Windows 10 Home 21H2 x64

CPU AMD Ryzen 5 1600 @ 3.9GHz

RAM 16GB 3000mhz cl16

GPU Nvidia GTX 1060 6GB

HDD 2TB 7200RPM

5. Results

The conventional and ideal approach when evaluating the performance of the runs would have been to use last year's test collection. However, since we did not have access to last year's corpus we have decided to use this year's test collection to evaluate our systems, using a *qrels* file containing relevance feedback manually performed by us.

The *qrels* file has been built by gathering, for each of the runs performed, the top 5 ranked documents for each topic.

Table 1
NDCG@5 and setup for single runs

#	NDCG@5	num_q	RF	Stoplist	Filter	Stemmer	Similarity	Weights	Reranking
1	0.3830	50	False	lucene	False	None	BM25	[1,1]	False
2	0.3756	50	False	lucene	False	None	LMD	[1,1]	False
3	0.3313	50	False	lucene	False	None	TFIDF	[1,1]	False
4	0.4140	50	False	smart	False	None	BM25	[1,1]	False
5	0.4258	50	False	terrier	False	None	BM25	[1,1]	False
6	0.4366	50	False	kueristop	False	None	BM25	[1,1]	False
7	0.4548	50	False	kueristopv2	False	None	BM25	[1,1]	False
8	0.4015	48	False	lucene	True	None	BM25	[1,1]	False
9	0.4759	41	False	kueristop	True	None	BM25	[1,1]	False
10	0.4823	48	False	kueristopv2	True	None	BM25	[1,1]	False
11	0.2634	50	False	kueristopv2	False	None	BM25	[0,1]	False
12	0.3654	50	False	kueristopv2	False	None	BM25	[1,0]	False
13	0.4525	50	False	kueristopv2	False	None	BM25	[1,2]	False
14	0.4674	50	False	kueristopv2	False	None	BM25	[2,1]	False
15	0.4873	50	False	kueristopv2	False	Porter	BM25	[1,1]	False
16	0.8549	50	True	kueristopv2	False	False	BM25	[1,1]	False
17	0.8552	50	True	kueristopv2	False	Porter	BM25	[1,1]	False
18	0.5867	48	False	kueristopv2	True	None	BM25	[1,1]	True
19	0.5392	50	False	kueristopv2	False	None	BM25	[2,1]	True
20	0.5714	50	False	kueristopv2	False	Porter	BM25	[1,1]	True
21	0.8606	50	True	kueristopv2	False	False	BM25	[1,1]	True
22	0.8323	50	True	kueristopv2	False	Porter	BM25	[1,1]	True

Table 2
NDCG@5 and setup for rrf runs

# fused	NDCG@5	Reranking
10,14,15,16,17	0.7521	False
10,14,15,16,17	0.7450	True

The runs' performance has been evaluated using *trec_eval*, the key measures considered are *NDCG@5*, the official measure used by CLEF to rank runs, and *num_q*, the number of topics retrieved (since some runs retrieved no documents for some of the topics).

All the runs, their characteristics and key measures are reported in Table 1 and 2. The five runs with their number in bold are the five submitted runs.

All the runs are performed on indexes obtained using Standard tokenizer and Lowercase filter, except for indexes used in runs obtained using Relevance Feedback, which use Letter tokenizer instead; this is because some of the tokens obtained using standard tokenizer were written in a format that caused errors when used as query (e.g. "text:text:text" would be a token that caused errors).

- The runs 1 to 3 compare BM25, Dirichlet and TFIDF Similarity as scoring functions, using lucene stoplist. The run using BM25 was the best performer, so we decided to use this Similarity for all the other experiments.

- Runs 1 and 4 to 7 compare different stoplists, in particular we compared lucene, smart and terrier stoplists and our own custom stoplists kueristop and kueristopv2; the results show that among the "generic" stoplists the larger ones have a bigger impact, but custom stoplists bring to even better improvements, with kueristopv2 being the best.
- We then wanted to assess the impact of filtering the runs by all the terms in the object field. Runs 8, 9 and 10 are performed adding the filter to the setup of runs 1, 6 and 7. Run 9 only retrieved documents for 41 topics, as 9 topics contain, in the ojects field, terms that are in the stoplist (and therefore are in the index); runs 8 and 10 retrieve documents for 48 queries, because lucene and kueristopv2 contain the terms "the" and "in", which again are in the objects field for two queries. The runs with filtering have a better *NDCG@5* score compared to runs without, however they retrieve less topics. Retrieving no documents for some topics make us assess these runs as worse performing compared to the ones without filtering. Moreover the improvement in *NDCG@5* score could be caused in part by the lack of these topics, as the system could have worse performance for these topics compared to the others. Despite having worse results when taken singularly, runs using filtering can be used to improve other runs by using *RRF*.
- Runs 11 to 14 use the same setup as the current best performing run, 7, changing the weight of Contents and DocT5Query fields respectively. When searching on a single field (weight 0 on the other field) the score is much worse, increasing the weight of DocT5Query field slightly worsens the score, increasing the weight of Contents field instead improve the score.
- Run 15 adds to the setup of run 7 a stemmer, specifically Porter stemmer; this addition brings to a good improvement in performance.
- Runs 16 and 17 instead are performed using Relevance Feedback, respectively without stemmer and with Porter stemmer; These runs have an NDGC@5 score incredibly higher than the previous ones, this however is due to using the same collection, and in particular the same *qrels*, to obtain the RF runs and to score its performance. To have a more reliable assessment of performance we could have done the search on a index built removing documents present in the qrels file. However, while this would have prevented the overfitting problem, we still couldn't have directly compared results to other runs; in fact, the documents in the *qrels* file, being the top documents retrieved, should be the most relevant, which mean we should have expected worse results by the runs performed when removing the documents from the collection.
- The first *rrf* run is obtained fusing a mixture of well performing and slightly different runs: 10, 14, 15, 16 and 17. It presents a very good NDCG@5 score, but since it uses *RF* runs the score is not reliable as these runs also may contain overfitting.
- Runs 18 to 22 and the second *rrf* run are obtained by applying reranking to the runs above (10, 14, 15, 16 and 17 and their fusion). Comparing to their non-reranked respectives we can see that results on RF and rrf runs are mixed, but again not the most reliable because of previous overfitting; on the other three runs instead reranking offers a really great improvement in performance.

6. Conclusions and Future Work

We managed to test out different techniques and tools studied in lessons, to discover new ones on our own and experiment first hand their impact in a "real world" application.

We managed to improve substantially the performance of the runs compare to the initial lucene baseline, with an increase in score of over 50% when considering our best performing non-overfitted run.

The greatest impact we noticed, except for the use of RF (that as explain we can't quantify), comes from reranking, the use of a stemmer and a stoplist customized to our corpus.

This is remarkable also because, due to the lack of access to last year's corpus, it wasn't possible for us to perform any fine-tuning.

Having access to such test collections would allow us for example to fine tune BM25 parameters, the field weights, the boosts for terms in RF, we could experiment with many more stoplists and stemmers. As an example, a run implementing Porter stemmer (or a different stemmer), fine tuned weights with the Contents field having more weight than the DocT5Query one would probably best all the other single runs, but the extra time it took us to also manually assess documents proved to be a strong limiting factor in the expansion of our experiments.

In future works it would be interesting to experiment with other "classic" method, for example using singles, but mostly with machine learning and deeplearning techniques, that have become the standard in the last decade of information retrieval; it would also be interesting having the chance to work with data in formats different than full-text, with the addition of metadata (for example in this case, since the corpus was created crawling the web having access to metadata from the webpages would have presented new opportunities).

References

[1] G. V. Cormack, C. L. A. Clarke, S. Büttcher, Reciprocal rank fusion outperforms condorcet and individual rank learning methods (2009). URL: https://plg.uwaterloo.ca/~gvcormac/cormacksigir09-rrf.pdf.

[2] Project debater for academic use, n.d. URL: https://early-access-program.debater.res.ibm.com/academic_use.